Compilation Switch Options

The compilation process and the resulting program can be modified by switches appearing in the PCL command line. Switches are a single alphabetic character following the '/' (slash) marker, as in the command line MAIN/D/S.

The complete set of compilation switches appears below, followed by a detailed description of each switch.

/C	Compile only	Complete compilation but no execution
/D	Debug	Debugger compilation
/E	External	External module - implies /O
/F	Fast reals	Generate calls rather than traps
/I	Improver	Branch/jump resolution
/L	Listing	Produce compilation listing
/L:n	Listing	Specify listing page length
/M	Macro	Partial compilation to assembler .MAC
/N	Nolist	List errors only
/O	Object	Partial compilation to linker .OBJ
/P	Profile	Profiler compilation
/Q	Quick	Uses fast MAC.SAV assembler
/S	Source	Include source lines (modifies /D/P/M)
/X	eXtend	Extended precision Reals (15 digits)

Listing Control Switches (/L, /L:n, /N)

The /L switch directs the compiler to produce a listing. The /L:n switch indicates that the listing is to be in pages of N lines each. The /N switch directs the compiler to list only lines in error. The /L and /N switches are related to the $L+ and $L- embedded switches.

Partial Compilation Switches (/C, /M, /O)

These switches interrupt the compilation process when intermediate results are desired.

The /M switch performs only the first compilation step, resulting in an assembler source translation of the Pascal program. The assembler source file has the extension .MAC. The /S switch is recommended with /M and will include the Pascal source lines as comments.

The /O switch performs the compilation and assembly steps, producing a relocatable object file suitable for input to the Linker or Librarian. The /O switch is implied by the /E (External) switch.

The /C switch performs the entire compilation process, but does

not execute the resulting program. The program may then be run with the R or RUN commands.

Real Arithmetic Switches (/X, /F)

The /X switch causes the compiler to use extended precison for values of type Real. All Real values are extended -- it is not possible to mix normal and extended precision values. The /X switch is related to the $X embedded switch. See the section on Extended Precision.

The /F switch is of limited utility. On processors lacking both FPP and FIS floating point hardware, Real operations are normally performed by trapping each FIS instruction and simulating its effects. The trapping process requires some overhead, but is compact. The /F switch causes the compiler to generate subroutine calls rather than simulating FIS instructions, which is faster but requires an extra word for each floating point instruction.

Debugger Switch (/D)

The /D switch indicates a Debugger compilation. This switch causes generation of a symbol table file and, if /S is also present, a listing file. The /D switch also causes generation of code to identify each procedure and statement to the interactive debugger. The /D switch is related to the $D+ and $D- embedded switches. See the section on the Debugger.

Profiler Switch (/P)

The /P switch causes the inclusion of code for performance measurement, and generates a symbol table file and, if /S is also specified, a listing file. The Profiler uses the Debugger interface code, so that /P cannot appear with /D. The /P switch is related to the $D+ and $D- embedded switches. See the section on the Profiler.

Source Mode Switch (/S)

The /S switch performs two distinct functions. When used with the Debugger (/D/S) or Profiler (/P/S), it enables the source program mode of operation and connects the actions of the Profiler and Debugger to the source text of the program.

When used with the Macro switch (/M/S), the generated assembler language translation will include the Pascal source lines embedded as comments within the assembly file. This use of the /S switch is related to the $S+ and $S- embedded switches.

External Module Switch (/E)

The /E switch indicates an external module compilation. This causes the outermost procedures and functions to be identified to the Linker with global entry names. An external module can include global declarations, procedures, and functions but is not required to include a main control section. The /E switch is related to the $E embedded switch. See the External Module section.

Branch/Jump Improver Switch (/I)

The /I switch inserts an additional step into the compilation process, a branch/jump resolver (IMP). IMP replaces branch/jump combinations with a single conditional branch where possible. The Improver runs slowly, but typically reduces program size by 6 to 8 percent. IMP is recommended for compilation of debugged production programs.

Fast Assembler Switch (/Q)

The /Q switch causes the compilation process to use the MAC.SAV assembler instead of the MACRO assembler. MAC.SAV is a single pass assembler which has no macro or local symbol capabilities, but it assembles compiler output in about one third of the time required by MACRO.

Embedded Switches

Embedded switches provide control of compilation options within the Pascal source program. Embedded switches have the form of a Pascal comment beginning with a dollar sign ($), followed by a single uppercase alphabetic character and possibly a plus or minus sign, as in (*$L+*). Several of the embedded switch functions can also be provided by compilation switches. Embedded switches have the advantage that once included in a program, they cannot be accidently omitted from a compilation.

The complete list of embedded switches below is followed by a more detailed description of each switch function. In general, a plus sign enables a particular function, and a minus sign disables it. Switches which are initially enabled are marked with [+]; switches marked [MBF] 'must be first' -- they must appear before any Pascal code.

$A-,$A+	Array bounds	Include array bounds check [+]
$C	Code insert	See the Embedded Assembly Code section
$D-,$D+	Debugger	Include debugger interface
$E-,$E+	External	External module compilation
$F-,$F+	Fast FPP	Enable floating point calls
$L-,$L+	Listing	Source lines in listing [+]
$S-,$S+	Source mode	Source lines in assembly
$T-,$T+	sTack check	Include stack overflow check [+]
$X	eXtend	Extended precision reals [MBF]

Error Checking Switches ($A, $T)

The $A switch controls the generation of code to check array references and ensure that the index is within the bounds of the array. Bounds checking is initially enabled; the $A- switch will disable checking. If enabled, each bounds check requires 8 words.

The $T switch controls stack overflow checking, and is initially enabled. Stack overflow is possible upon entry to any procedure or function block. This switch can be disabled with $T-, resulting in small savings of memory (2 words per procedure).

Debugger/Profiler Switch ($D)

The $D switch controls the interface code to the Debugger and Profiler. If enabled, each statement and procedure includes instructions to call the Debugger or Profiler. These instructions require 1-3 words per statement (1 word for statements 1-255 of each procedure, 2 words otherwise, and an additional word if /S source mode is enabled). In large programs, one may choose to disable debug interface code generation for sections which are known to be correct.

External Module Switch ($E)

Enabling the $E switch causes global procedures and functions to be labeled as external entry points in the relocatable object file. A main program section encountered when the $E switch is enabled is ignored. See the External Module section.

Real Arithmetic Mode Switches ($X, $F)

The $X switch enables extended precision (15 digit) real arithmetic. If present, the $X switch must precede any Pascal code. Note that it is not possible to mix normal and extended precision in one program, so that each module in separate compilations must be compiled with the same precision. See the Extended Precision section.

The $F switch is useful only on processors which lack FIS and FPP hardware for floating point calculations. On these processors, floating point instructions are normally trapped and simulated. The $F switch instead causes direct subroutine calls to floating point routines, saving about 0.2 milliseconds per floating point instruction at the cost of an extra word.

Listing Control Switch ($L)

The $L switch controls the appearance of lines in the program listing file. If enabled, all program text will appear in the listing. If the $L switch is disabled, only lines in error and error messages will appear.

Source Mode Switch ($S)

Enabling the $S switch causes the Pascal source lines to appear in the compiler assembly output as comments. This makes it easier to determine the code generated for each statement.

I/O Control Switches

The Reset() and Rewrite() standard procedures accept additional arguments specifying a Filename of an external file, and a DefaultName with default fields of the filename. These arguments can also include I/O control switches which give explicit control of the operating system interface details.

The I/O switches appear in the Filename or DefaultName parameters as in this example:

```
Rewrite(F,'data.dat/seek/span/size:12.');
```

A complete list of I/O switches appears below, followed by individual details. All switches may be abbreviated to the first two letters.

/buffersize:n	Allocate N bytes for buffer
/go	Allow programmed error handling
/odt	Single character terminal input
/seek	Direct-access file
/size:n	File storage allocation
/spanned	Records span block boundaries
/temporary	Temporary file

/BUFFERSIZE:n Switch

OMSI Pascal-1 normally allocates the minimum space required for a file buffer, which is usually 512 bytes but is dependent on device and file characteristics. More efficient I/O transfers can be performed at the cost of additional memory. The /Buff:n switch specifies the storage to be allocated to a file buffer. The size value is a decimal number if terminated with a period, otherwise octal.

/GO Switch

I/O transfer errors are normally fatal and cause immediate program termination. The /Go switch indicates that transfer errors on the specified file are non-fatal and allow program execution to continue. Use of this switch implies that the programmer accepts responsibility for checking the RT11 I/O status code after each I/O operation. The error code for the previous I/O transfer error is available in the byte at address 52B.

/ODT Switch

The /ODT switch derives its name from the ODT Debugger (Octal Debugging Technique), which is driven by single character commands. The /ODT switch is used with keyboard files, and indicates that each character read from the file is to be processed immediately without waiting for a carriage return or other action character. The /ODT switch also disables the normal one character buffer effect of the Read() standard procedure.

The rubout and ctrl/U keyboard editing capabilities are not effective on terminals open with /ODT.

/SEEK Switch

The /Seek switch performs two functions: it enables the use of the direct-access Seek() procedure, and it permits both read and write access to the file variable so that record updating may occur.

/SIZE:n Switch

The /Size switch used in the Rewrite() procedure specifies the space to be allocated for the file. The size of the file is given in blocks of 512 bytes, and is a decimal number if terminated by a period, and octal otherwise.

/SPANNED Switch

In files created or accessed by OMSI Pascal-1 programs, records are normally 'blocked'. This means that an integral number of records are stored in one disk block of 512 bytes, with any remaining storage in that block being unused. The /Spanned switch causes records to be packed more efficiently, with records spanning from one disk block to the next. This requires additional buffer memory which is automatically allocated, and some additional computation.

Spanned and blocked files are not generally compatible. Files created with /Spanned should be read using the same switch.

/TEMPORARY Switch

This switch is used in Rewrite() to indicate a temporary file which will be deleted on termination. No filename is needed if this switch appears.

The Profiler

The Profiler is a program measurement tool which can be used to identify the sections of a program that can be most effectively optimized. Empirical measurements show that typical programs consume a large fraction of their computation time in a small portion of the program code ("90% of the time in 10% of the code"). The Profiler counts the actual number of times each statement is executed and each procedure is activated, and displays this information either in the program listing or in a tabular form.

The /P switch appears in the compilation command to invoke the Profiler. The /S switch is recommended in addition for more convenient display of the profile information.

When the Profiler begins execution, it will ask for the program name. The Profiler uses the symbol table and listing files produced by the compiler to identify procedures and statements in the program. The symbol table file normally has the same name as the program and the extension .SYM, and the listing file normally has the extension .LST. The Profiler will ask for the correct filenames if the normal files are not available.

The Profiler will then ask for the desired destination of the profile information. The profile will be written to the specified file with the default extension .PRO. This should be a permanent file (disk or hard copy device), as the Profiler requires roughly a factor of fifty performance overhead while gathering information.

The program being measured will then execute normally, although somewhat more slowly. Upon normal termination, or any fatal error, or ctrl/C interrupt, the profile information will be written to the specified file.

The first section of the profile is the Procedure Reference Profile, which lists each referenced procedure and function with the count of calls on that procedure. The second section is the Statement Reference Profile, displayed in tabular format. If the /S (source) switch is specified, this section displays the program listing with an additional column containing the reference count for each line.

The Profiler is limited in several respects: only the first 100 statements in each procedure will be counted, and a maximum of 40 procedures and functions can be profiled. The $D- and $D+ embedded switches can be used to selectively enable and disable profiling.

Example:

```
PRIMES                    OMSI Pascal V1.2E RT11 26-Mar-80  0:49 Site #1-1   Page 1
Oregon Software  2340 SW Canyon Road  Portland, Oregon 97201  (503) 226-7760

Line  Stmt Level  Nest   Source program

   1                     program Primes;(* Author: N. Wirth *)
   2                     constN=2500;(* first 2500 Primes *)
   3                     typeIndex=1..N;
   4                     varX,Square: integer;
   5                     I,K,Lim: Index;
   6                     Prime: Boolean;
   7                     P: array[Index] of integer;
   8                     V: array[1..100] of integer;
   9                     begin
   1   10    1    1    1   P[1]:=2; write(2); X:=1; Lim:=1; Square:=4;
   1   11    6    1    1   for I:=2 to N do begin
2499   12    8    1    3     repeat
11153  13    9    1    4       X:=X+2;
11153  14   10    1    4       if Square<=X then begin
  35   15   12    1    6         V[Lim]:=Square;
  35   16   13    1    6         Lim:=Lim+1; Square:=P[Lim]*P[Lim];
  35   17   15    1    6         end;
11153  18   16    1    4       K:=2; Prime:=true;
11153  19   18    1    4       while Prime and (K<Lim) do begin
94012  20   20    1    6         if V[K]<X
28669  21   21    1    7           then V[K]:=V[K]+P[K];
94012  22   22    1    6         Prime:=(X<>V[K]); K:=K+1;
94012  23   24    1    6         end;
11153  24   25    1    4     until Prime;
 2499  25   26    1    3     P[I]:=X; write(X);
 2499  26   28    1    3   end;
    1  27   29    1    1 end.
```

Format and Cross-Reference (PASFMT)

The PASFMT utility supplied with OMSI Pascal-1 will automatically reformat a Pascal source program, adjusting indentation and partitioning statements so that a program listing reflects the program structure. The PASFMT program can also provide a cross-reference index of a Pascal source program showing block calls, nesting, and identifier references.

The PASFMT command line can contain one or two output files, an input source file, and several optional switches. Run PASFMT as follows:

```
.R PASFMT
PASFMT V2.0 (10Dec79)
*(Format),(Crossref)=(Source)/switches
```

The Format output file is the formatted source program. Several switches select token translation options:

/L	Lowercase	Lowercase identifiers, uppercase keywords
/M	Mixedcase	Unchanged identifiers, uppercase keywords
/U	Uppercase	All letters uppercase

The Crossref output file (if specified) normally contains the program listing with line and page numbers, followed by the procedure call and nesting index. Two switch options apply to the cross reference:

/C	Crossref all	Cross reference all identifiers
/N	No listing	Produce only crossref index

The /C switch may be used only for source programs of moderate size, due to memory limitations.

The Improver (IMP)

The utility program IMP decreases the size of the object code produced by OMSI Pascal-1 by replacing branch/jump combinations with single branches when possible. IMP will reduce the generated code by roughly 5 to 8 percent.

IMP is included in the compilation process by the /I compilation switch, as in the command line TEST/I.

Note that IMP runs quite slowly and therefore is recommended for use only on completely debugged production programs.

Extended Precision

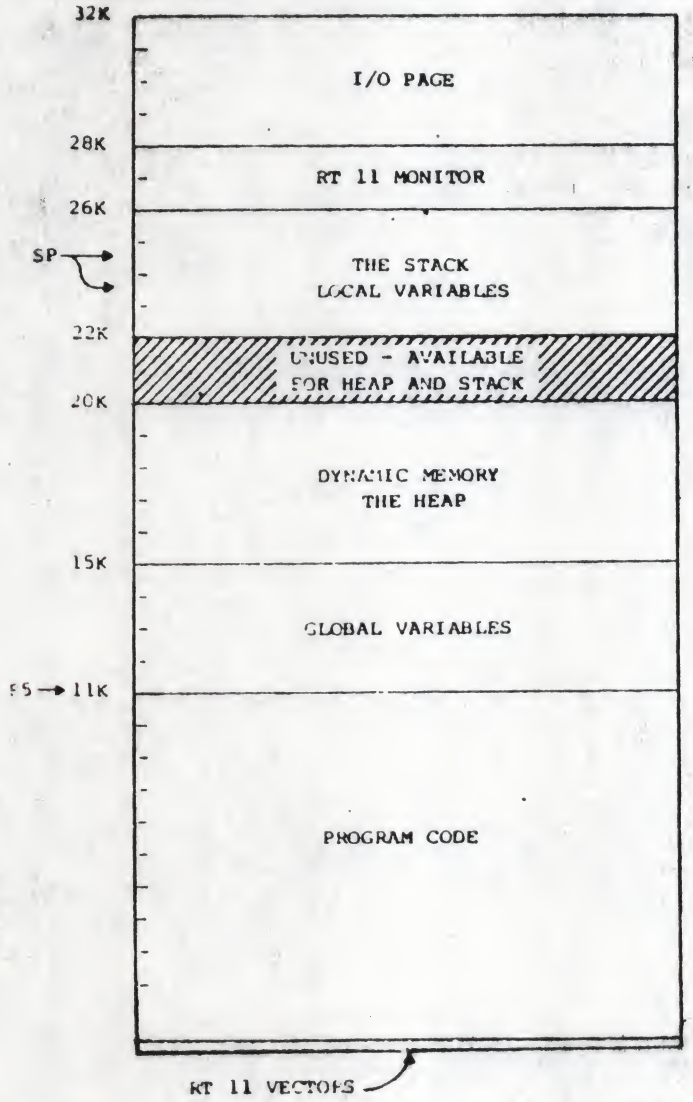
Values of type Real are normally stored in the PDP-11 single precision format, which requires 2 words of storage per value and offers 7 decimal digits of precision. The /X compilation switch or the $X embedded switch cause all Real values to have extended precision. Extended precision values each occupy 4 words of storage, and provide 15 digit precision in all real calculations, including the transcendental functions.

Extended precision applies to all Real values in a program -- it is not possible to mix normal and extended precision variables. All external modules must be compiled with the same precision as the main program, even if no Real variables are present.

Compared to normal precision Real variables, extended precision variables require twice the storage. The effect on computation time is dependent on the processor hardware: the FPP floating point processor provides hardware support for extended precision with a slight performance penalty (30%); processors with the FIS floating instruction set will experience the most severe relative performance penalty (20 to 1), because FIS offers no extended support and extended calculations are performed entirely in software; processors lacking any floating point hardware can expect a performance penalty of about 100%.

Runtime Memory Organization

A PDP-11 program has an address space of 32,768 words, or 32KW (1KW is 1024 words). The figure below shows this address space as it might be allocated for a typical program of moderate size.



This figure represents a snapshot taken during program execution, illustrating the partitioning of available memory. Each partition is described in the following sections.

RT11 Vectors and Communication

The RT11 Vector partition begins at address zero and occupies the first 256 words of all programs. This area contains interrupt vectors and RT11 status indicators. This area is also used for communication between the Pascal program and other programs called by chaining.

Program Code

The Program Code partition contains all of the instructions of the user program, including overlays, external modules, and routines from the runtime library. It is allocated memory adjacent to the RT11 Vector partition. The size of this partition depends entirely on the size of the user program.

Global Variables

The Global Variable partition contains all of the program's global variables - those defined in the outermost, or main, block of the program. This partition is fixed in size during program execution.

Register 5 (R5) points to the base of the Global Variable partition and is used for access to global variables.

Dynamic Memory - The Heap

The Dynamic Memory partition contains I/O control blocks and buffers, and variables allocated by the New() procedure. The Heap is unique in that it is not allocated any memory initially, but instead expands as necessary. The Heap is allocated adjacent to the Global Variable partition, and may grow on demand to the upper limit imposed by the Stack partition. The error message 'New() exceeded memory' indicates total exhaustion of memory resources.

Local Variables - The Stack

The Stack partition contains all variables local to inner blocks of the program, and is also used for temporary calculations, parameter passing, and subroutine return information. At the time a block is entered, a stack frame is created which contains all information local to that block. Stack frames are created and released in a purely nested fashion. See below for a detailed description of a stack frame.

The current Stack frame is always pointed to by the Stack Pointer (SP, register R6), which initially points to the top of the Stack partition. As nested Stack frames are allocated, the Stack Pointer decreases in value (points to lower addresses). If the Stack partition is too small, the Stack Pointer will eventually overrun the Heap partition and cause the 'Stack exceeded memory' error.

RT11 Resident Monitor

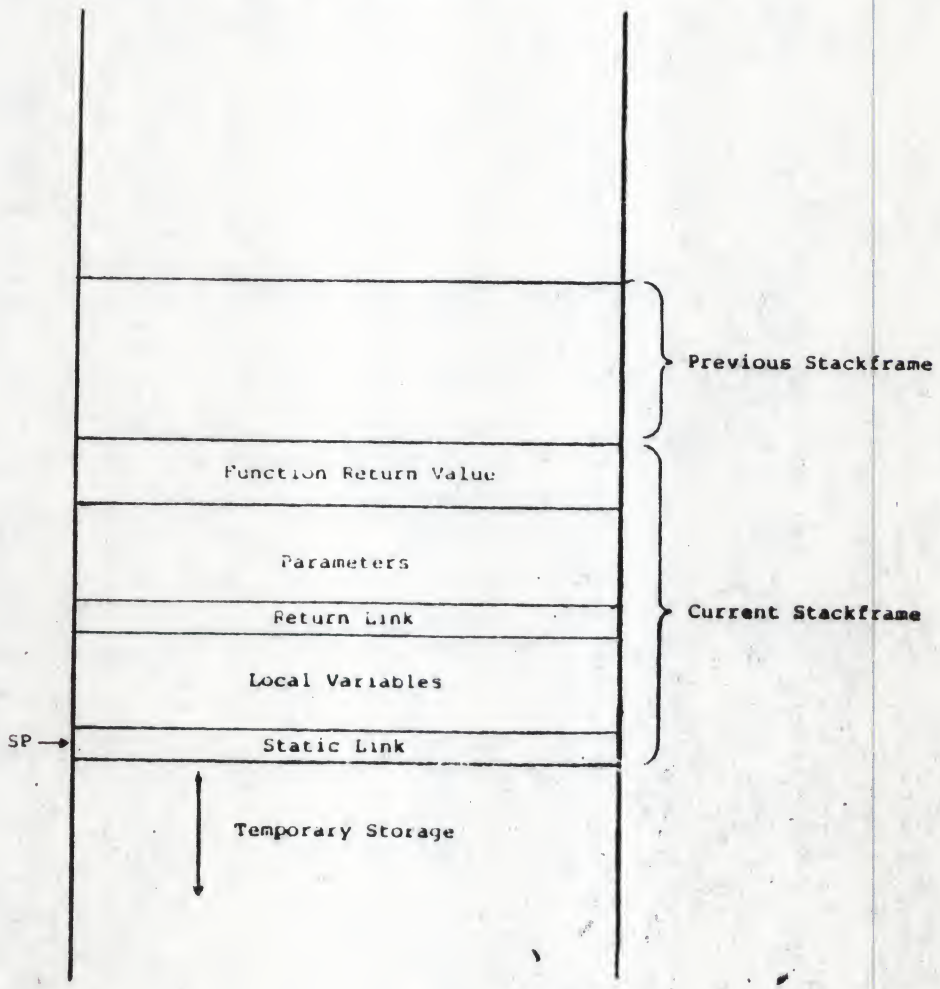
This area is at the high limit of available physical memory, and contains the fixed RT11 Resident Monitor, the User Service Routine (USR) if it is set NOSWAP, and any device handlers loaded with the LOAD command.

I/O Page

The I/O Page of address space contains all device status and command registers and internal processor registers.

A Closeup of the Stack Partition - The Stack Frame

The Stack partition is entirely composed of Stack Frames. A Stack Frame is created during entry to every block (excluding the main program block), and is released when the block is exited. The following diagram illustrates the possible components of a Stack Frame. Most of these components are optional -- only the Return Link is required in every Stack Frame.



Function Return Value

This field is present in Stack Frames associated with function blocks, and holds the value to be returned by the function. Its position at the bottom of the Stack Frame allows it to be 'popped' from the stack when control returns to the caller of this block.

Parameters

The Parameters field contains either parameter values or their addresses. Blocks without parameters do not have this field in their Stack Frame(s).

Return Link

This field is the subroutine return address, where control is transferred on exit from this block.

Local Variables

This field contains all local variables for this block. It does not appear for blocks without local variables.

Static Link

The Static Link appears only in blocks which are lexically enclosed by other procedure or function blocks. The Static Link is used for references to intermediate level variables in the enclosing block(s). It points to the base of the Stack Frame of the latest invocation of the immediately enclosing procedure or function block, and it is the first link in the Static Link chain.

The Stack Pointer (SP) is also used for transient temporary storage, as in interrupts and Pascal library calls, and each For statement requires 3 words of temporary stack storage during its execution.

Dynamic String Package

A package of procedures and functions for dynamic string processing is supplied with OMSI Pascal-1 V1.2, and is stored in the file STRING.PAS. The package is written in Standard Pascal, and allows programs using strings to be moved to other Pascal implementations.

Strings are stored as a record structure with a fixed maximum number of characters (normally 100 but easily changeable), and an integer marking the current length of the string.

```
type String = record
        Len: Integer;
        Ch:  packed array[1..StringMax] of Char;
        end;
```

The capabilities provided are:

Len(S) - returns the current length of string S;

Clear(S) - initializes string S to empty;

ReadString(F,S) - reads a value for string S from the text file F. The string is terminated by Eoln(F) and a Readln(F) is performed. String overflow (a string longer than StringMax) results in truncation.

WriteString(F,S) - writes the string S to the text file F. The same effect can be achieved by passing the parameter S.Ch:S.Len to Write(), as in Write(F,'S=',S.Ch:S.Len).

Concatenate(T,S) - appends string S to the target string T. The resulting value is string T. Overflow results in truncation to StringMax characters.

Search(S,T,Start) - searchs string T for the first occurrence of string S to the right of position Start (characters are numbered beginning with one). The function Search() returns the position of the first character in the matching substring, or the value zero if the string S does not appear.

Insert(T,S,Start) - inserts the string S into the target string T at position Start. Characters are shifted to the right as necessary. Overflow produces a truncated target string. A Start position which would produce a string which is not contiguous has no effect.

The Start and Span parameters in the Substring and Delete procedures define a substring beginning at position Start (between characters Start-1 and Start) with a length of Abs(Span). If Span is positive, the substring is to the right of Start, and if negative, the substring is to the left.

Delete(S,Start,Span) - deletes the substring defined by Start, Span from the string S.

Substring(T,S,Start,Span) - the substring of string S defined by Start, Span is assigned to the target string T.

External Modules

External modules allow several program sections, each containing at least one procedure, function, or main program, to be compiled independently and combined at link time. External modules may be combined into module libraries to simplify handling of common routines. The external module interface also allows inclusion of modules written in other languages, such as FORTRAN and MACRO.

The EXTERNAL directive is used to reference a procedure or function in an external module. The declaration of an external procedure or function contains the procedure or function name and parameters, followed by the directive EXTERNAL (similiar to FORWARD). The procedure or function body does not appear in the program unit referencing the external routine.

The FORTRAN directive replaces EXTERNAL to reference external routines written in FORTRAN or MACRO. The FORTRAN directive causes the generation of a PDP-11 standard calling sequence (the Pascal calling sequence places parameters on the stack, while the FORTRAN sequence points R5 to a list of parameters).

The /E compilation switch and the $E embedded switch are used to create modules which can be referenced by EXTERNAL directives. When the $E switch is enabled, each global procedure and function declaration causes an external (global) symbol to be defined. These global symbols are matched at link time to the global references created by the EXTERNAL directive.

The external reference symbols are composed of the first six characters of the external procedure or function identifier, and must uniquely identify the external routine. Duplication or overlap of external symbols results in the Link error 'Multiple definition', while a missing module results in the 'Undefined global' error message.

Several cautions should be observed when using EXTERNAL and FORTRAN directives. Parameters to external modules cannot be checked by the compiler for type conformance, so an accidental type mismatch may cause entirely unpredictable results. The FORTRAN directive causes only generation of the proper calling sequence; it does not link or initialize the FORTRAN I/O system.

External modules may reference global (static) variables, which are shared by all of the modules composing a program. If all modules (including the main program) are compiled with the same global variables, the effect is as if all modules were compiled together. The compiler cannot verify type conformance of global data.

When combining modules to form libraries, remember that all procedures and functions from a compilation form a single module, and cannot be individually selected from the library. The module

name is taken from the first six characters of the program identifier (in the program heading).

The Linker, Librarian, and Overlays

Object modules produced by the Pascal compiler using the /O or /E compilation switches are compatible with object modules produced by the MACRO assembler, FORTRAN compiler, and other RT11 system utility programs. The Linker (LINK) can be used to produce overlaid executable programs, allowing much larger programs. The Librarian (LIBR) is used to build libraries of object modules for more convenient handling. Some highlights of Linker and Librarian capabilities are covered here -- see the RT11 System User's Guide for complete details.

To run the Linker, give the command

```
.R LINK
*
```

('*' indicates the Linker is waiting for a command). The first command line can include the output file, a map file if desired, and up to six input files. The file PASCAL.OBJ, which contains the Pascal runtime library, must appear in every Pascal link procedure. The /C switch (continue) allows commands to be continued onto the next line.

```
*OUT,MAP=MAIN,SUB1,LIB1/C
*PASCAL
*↑C
```

The overlay facilities of the Linker are selected using the /O:N switch, where the parameter N indicates the overlay region number. Sets of modules which are allocated to the same region will be overlaid against other modules in the same region, with only one set of modules per region actually in memory at any one time.

The following sequence links a main program, an external module, and the Debugger into an overlaid executable file. The main program, external module, and the Pascal library are not overlaid. Debugger modules A and B do not call each other, and are overlaid in region 1. Debugger modules 0-9 do not call each other, and can be overlaid in region 2.

```
.R LINK
*PROG=MAIN,SUB1,PASCAL/C
*P:A/O:1/C
*P:B/O:1/C
*P:0/O:2/C
*P:1/O:2/C
*P:2/O:2/C
*P:3/O:2/C
```

```
*P:4/O:2/C
*P:5/O:2/C
*P:6/O:2/C
*P:7/O:2/C
*P:8/O:2/C
*P:9/O:2
*↑C
```

The Librarian combines relocatable object modules to form object libraries. These libraries may be included as input to the Linker, which will select only those modules which are needed by the program being linked. Note that a module always consists of the entire set of procedures and functions from its compilation -- individual procedures cannot be selected from a module.

For example, the string package STRING.PAS can be edited to form 9 modules, with each module containing one procedure or function. The 9 modules can then be compiled, and combined into a library as follows.

```
.R LIBR
*STRING=LEN,CLEAR,READS,WRITES,CONC/C
*SEARCH,INSERT,DELETE,SUBS
*↑C
```

Embedded Assembly Code

PDP-11 assembly code can be embedded within an OMSI Pascal-1 program at any point where a comment might appear. Embedded assembly code takes the form of a special comment beginning with the embedded switch $C, as in the comment

```
{$C MOV %0,-(%6) }
```

The assembly code section extends to the closing comment brace (this closing brace cannot appear in an assembler comment). Any of the capabilities of the MACRO assembler may be used.

The OMSI Pascal-1 compiler scans the embedded assembly code and replaces tokens within the code which correspond to certain classes of Pascal identifiers. This provides simplified access to Pascal data and control structures. However, the programmer is required to have some understanding of the internal structures. See the section on Runtime Memory Organization, and examine the code produced by the compiler. Constant identifers appearing in assembly code are replaced by their defined values. Variable identifiers are replaced by the numeric offset from the appropriate base pointer. For global variables, the base pointer is Register 5 (R5); for local variables, the stack pointer (SP) is the base. For example, to swap the halves of a local integer variable I, the code would be

```
{$C SWAB I(SP) }
```

and to assign the constant Ten to the global variable Count one can write

```
{$C MOV #TEN,COUNT(R5) }
```

Any temporary stack usage is not recognized by the compiler, and must be included in indexed addressing of local variables.

Parameters of Pascal procedures and functions are treated as local variables, and are accessible in the same fashion. Internally, a Var parameter is the address of the actual parameter, so references to Var parameters must be indirect, as in

```
{$C MOV @VAR(SP),R0 }
```

Procedure and function identifiers are replaced by the internal label assigned by the compiler. To assign a value to a function, it is best to move the value to a local variable and then use a Pascal assignment statement to copy the value to the function.

The programmer is responsible for selecting the proper base register, as the compiler provides no error checking capability. Identifier substition is performed for all identifiers in these classes. This can cause problems if the programmer defines an identifier which corresponds to a MACRO operation, such as a constant named 'MOV'.

With one exception, registers R0-R4 are available within embedded code sections. The exception is With statements, each of which maintains a fixed address in a register. With register allocation is in the order R3,R2,R1,R0 and can be determined from the Pascal program. Registers R5 and SP must be preserved across the range of an embedded code section.

The default numeric radix of a Pascal-produced assembly code file is decimal, not the normal octal.

The System Error() Procedure

When a fatal runtime error occurs, the system procedure Error() is called with parameters describing the error and the system state. The Error() procedure is known by the global name ERROR, and may be replaced by a user-written external module of the same name. The external module must accept the parameters defined below.

```
type Class = (Fatal,IOError,Warning);
     Message = packed array[1..100] of Char;
procedure Error(
     ErrorClass: Class;
     ErrorNumber: Integer;
     ErrorMsgLength: Integer;
var ErrorMsg: Message;
var XFile: Text;
     IOStatus: Integer;
     UserPC: Integer;
     FilenameLength: Integer;
var Filename: Message;
)
```

The ErrorClass parameter indicates the type and severity of the error; Fatal and IOError are errors with no possible recovery, while Warning errors will recover automatically. The ErrorNumber indicates the exact cause of the error. ErrorMsgLength and ErrorMsg define the text of the printed error message normally displayed for this error. The XFile parameter identifies the file variable associated with this error, if any. IOStatus is the value of the RSTS/E I/O status word. UserPC is the program counter saved at this error, which can often be used to identify the program segment responsible for the error. Finally, FilenameLength and Filename describe the external name associated with the file variable XFile.

The possible courses of action available to the Error() procedure are very limited, as exiting from the Error() procedure normally results in termination. The program global variables are available and may aid in diagnosing the problem. The Error() procedure may provide operator interaction or recording capabilities beyond the normal messages to the terminal, and as a final resort may call on operating system facilites to 'chain' and restart the program or initiate another program.

Compiler Error Messages

',' used instead of ';'
8 or 9 in octal constant
Argument must be integer
Argument must be ordinal type
Argument must be real
ARRAY index out of range
ARRAY index type error
Bad ABS argument
Bad argument
Bad CASE label
Bad constant
Bad EXIT
Bad expression
Bad field list
Bad FILE name
Bad FOR statement
Bad FUNCTION name
Bad FUNCTION result type
Bad IN operands
Bad index type
Bad LABEL
Bad ORIGIN for variable
Bad parameter
Bad PROCEDURE name
Bad PROGRAM name
Bad READ statement
Bad RECORD
Bad scalar type
Bad SET element
Bad subrange
Bad TYPE
Bad TYPE specification
Bad variable list
Bad variant
Bad WITH statement
Bad WRITE statement
Boolean expression needed
Constant overflow
Don't repeat FORWARD parameter list
Duplicate CASE label
Duplicate field name
ELSE must be last in CASE
Expression too complex - out of registers
Expression too complex - out of registers (real)
Field list must be in parentheses
File variable missing
Format expression must be integer
FORTRAN must be VAR parameters
FUNCTION arg must be real or integer
FUNCTION argument missing

Illegal assignment
Illegal character
Illegal operator
Illegal type of operand
Improper symbol
Incompatible ARRAY type
Incompatible type
Invalid declaration, probably missing END
Invalid symbol
LABEL defined at wrong level
Label must be integer
LABEL not declared
LABEL redefinition
Local VAR definitions must precede PROCEDURE definitions
Missing ')'
Missing ')' at end of list
Missing '.' at end of program
Missing BEGIN
Missing END
Missing END in CASE
Missing field variable
Missing LABEL
Missing label definition
Missing operand
Missing operator
Missing semicolon
Missing UNTIL
Must be simple variable
NEW or DISPOSE arg must be pointer
Not implemented
ODD argument must be integer
Output file error
Source line too long
Strange '[' - bad SET or missing ARRAY definition
TEXT file expected
Too few arguments
Too many arguments
Too many errors in this line
Too many errors!
Too many levels
Too many symbols
Undefined FORWARD PROCEDURE or FUNCTION
Undefined operand
Undefined pointer base type
Undefined symbol
Unresolved forward type reference
WITH nested too deep

Runtime Error Messages

Bad filename
Can't Reset(output)
Can't Rewrite(input)
Compiler/library mismatch -- please recompile
Dispose(nil) attempted
Division by zero
Duplicate Dispose()
End of file
Exp() overflow
File not open
Floating point format error
Get() not allowed
I/O transfer error
Illegal value for integer
Integer overflow
Log() of zero or a negative number
New() exceeded memory
New() of zero length
No channels available
No file to Reset()
Overlay error
Put() not allowed
Real overflow
Reset() failure
Rewrite() failure
Seek() on sequential file
Seek() out of range
Set element out of range
Sqrt() of a negative number
Stack exceeded memory
Subscript out of bounds
Trunc/round overflow
Unexpected trap